PUBLIC

Code Assessment

of the Security Council AIP Smart Contracts

March 18, 2024

Produced for

ARBITRUM FOUNDATION

by

CHAINSECURITY

Contents

1 Executive Summary

Dear Arbitrum Foundation Team,

Thank you for trusting us to help Arbitrum Foundation with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of Security Council AIP according to Scope to support you in forming an opinion on their security risks.

Arbitrum Foundation implements an Arbitrum Improvement Proposal (AIP) that aims to increase the signature threshold of the non-emergency Security Council multisig on Abitrum One (`0xADd68bCb0f66878aB9D37a447C7b9067C5dfa941`) from 7 to 9 signatures. Moreover, a library for conditional updates of the constitution was implemented.

The most critical subjects covered in our audit are the functional correctness of the proposal and the correctness of the proposal with regards to lifecycle of a proposal in the arbitrum ecosystem. Security regarding all the aforementioned subjects is high.

The general subjects covered are access control, testing, documentation and specification. There was no end-to-end testing for the proposal flow. Security regarding all the rest of the aforementioned subjects is high.

In summary, we find that the codebase provides a high level of security.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	0

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the Security Council AIP repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	6 March 2024	1bc50c554192620a4e8b6cb741345ef478d8fc67	Initial Version
2	18 March 2024	d90249166f22be53b4b808007c5a704c87102dac	Final Version

For the solidity smart contracts, the compiler version `0.8.16` was chosen.

The following contracts are in scope under the `src/gov-actions-contracts/` directory:

- `AIPs/SCImprovementAIP/AIPIncreaseNonEmergencySCThresholdAction.sol`
- `governance/ConstitutionActionLib.sol`
- `governance/SetSCThresholdAndUpdateConstitutionAction.sol`

2.1.1 Excluded from scope

All the contracts not mentioned in scope are considered out of scope. The contracts under review are going to be executed on Arbitrum One. The difference in the semantics of the EVM opcodes that could be introduced by the Sequencer is beyond the scope of this audit. The contracts that are used during the lifecycle of the proposal e.g., L2ArbitrumGovernor, L1ArbitrumTimelock, etc are assumed to function properly. Issues related to the bridging mechanism used during the propagation of the proposal from Arbitrum One to Ethereum and vice versa are out of scope. Finally, the multisig is assumed to have been configured correctly to successfully execute the proposal.

2.2 System Overview

This system overview describes the initially received version (Version 1) of the contracts as defined in the Assessment Overview.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

Arbitrum Foundation offers an Arbitrum Improvement Proposal (AIP) as well as an implementing mechanism to increase the threshold of the non-emergency Security Council from 7/12 to 9/12. As this AIP changes the Arbitrum DAO constitution, it is considered as a constitutional AIP.

2.2.1 Security Council

Security council supervises a 12-members multisig wallet for non-emergency actions on Arbitrum One. If considered as routine operational (e.g., routine system upgrades), 7/12 approval suffices. This is mentioned in the DAO constitution:

...Performing any Emergency Action requires a 9-of-12 approval from the Security Council ... The Security Council may also approve and implement routine software upgrades, routine maintenance and other parameter adjustments in a non-emergency setting (such actions, "Non-Emergency Actions"), which require a 7-of-12 approval in order to take effect.

This AIP targets an increase of 7/12 to 9/12 for even non-emergency actions. Hence, it is treated as a constitutional AIP.

2.2.2 Proposal Lifecycle

Whether an AIP is constitutional or not, the governance process varies. In this report, we focus on the phases a constitutional AIP goes through.

Phase 1: Temperature check

Any new AIP should firstly be discussed at least one week in the Arbitrum Forum. After at least one week of discussion, AIP can be moved to an off-chain vote in the snapshot. To create a Snapshot poll, the author must hold at least 0.01% of the total “votable” tokens. To vote, members must hold or be delegated the ARB token. After one week, the poll is decided by the majority vote.

Phase 2: Formal AIP

The proposer can submit an on-chain vote. There is a 3-days window between submitting the proposal and the beginning of the voting.

Phase 3: On-chain DAO vote

By calling `L2ArbitrumGovernor.propose()`, voting begins and can go on for up to 16 days. For a constitutional AIP to pass, it should

1. receive more than 5% of all the votable tokens in favor
2. and, a simple majority of the votes in favor.

Phase 4: L2 waiting period

After a proposal is passed, it goes through an additional 3-day waiting period, to allow the affected parties to react to the proposal e.g., by withdrawing their assets to Ethereum before the proposal goes into effect. The proposal is enqueued by `L2ArbitrumGovernor.queue()`, and this function internally calls `schedule()` on the `ArbitrumTimeLock` contract, which queues the message for a 3-days interval.

Phase 5: L2->L1 messaging

After the `ArbitrumTimeLock` delay has passed, anyone can call `execute()` on `ArbitrumTimeLock`. This function calls the proposal target, which usually is an ArbSys precompile contract, with the proposal payload set by the proposer. This payload is used later. After the message lands in the L1 Outbox, anyone can process it. L2->L1 messaging takes a week (challenge period window), to ensure the challenge period is elapsed and the inserted state root from L2 to L1 is correct. Executing the message in L1 Outbox calls `L1TimeLock.schedule()`, which subsequently adds the message to a queue to be executed later.

Phase 6: L1 waiting period

An extra 3-days of keeping the message on L1, gives this opportunity to any pending transaction on the L1 to finalize. After this phase, any user can call `L1TimeLock.execute()`. Depending on the payload set in the Phase 5, either execution happens in L1 or a retryable ticket is issued in the L1 Inbox with the target being a contract on L2. In our case, this is `UpgradeExecutor` on Arbitrum One.

Phase 7: Enforcing the changes

Finally, changes outlined in the AIP are implemented. After the retryable ticket is received in L2, it is going to be handed over to the `UpgradeExecutor`. This contract makes a delegatecall to the `perform()` function of the proposal. In case of increasing the SC threshold, the following function is called:

```
gnosisSafe.execTransactionFromModule({
    to: address(gnosisSafe),
    value: 0,
    data: abi.encodeWithSelector(_IGnosisSafe.changeThreshold.selector, newThreshold),
    operation: OpEnum.Operation.Call
});
```

Moreover, the hash of the constitution is updated. The owner of the constitution contract is the `UpgradeExecutor`.

2.2.3 Module System of Safe

Even though it is beyond the scope of the review, we include some details regarding the implementation of the Safe multisig being used. Safe multisigs can specify the so-called modules. These are contracts that extend the functionality of the multisig. The `UpgradeExecutor` is such a module. `UpgradeExecutor` implements the `execute()` function which performs a delegate call on a target specified by its arguments. `execute()` can only be called by the executor role. `L1ArbitrumTimelock` alias on L2 has executor role. The 9 of 12 security council on L2 has also this role.

2.3 Trust Model and Assumptions

`UpgradeExecutor.ADMIN_ROLE`: admin of `UpgradeExecutor` can setup roles for other users, especially `EXECUTOR_ROLE`. Hence, fully trusted. `UpgradeExecutor.EXECUTOR_ROLE`: users holding this role are allowed to call `UpgradeExecutor.execute()`. These users must be fully trusted, otherwise, they can misuse the high privilege of calling `execute()`.

We assume that in each proposal:

1. the correct `UpgradeExecutor` is set and
2. the `UpgradeExecutor` must be added as a module to the `gnosisSafe` to accomplish the call to `perform()`.

The execution of the proposal depends on the liveness of the rollup and its ability to exchange messages with the L1. We assume that the system can always make progress and that a proposal will eventually be executed by the rollup no matter the state of the sequencer.

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	0

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	0
Informational Findings	1

- Typo in the Error Message Code Corrected

6.1 Typo in the Error Message

Informational Version 1 Code Corrected

CS-AFSCP-001

To check whether the new constitution was set correctly the following snippet is used:

```
require(constitution.constitutionHash() == newConstitutionHash, "NEW_CONSTUTION_HASH_SET");
```

The error message contains a typo. It should be `CONSTITUTION` instead.

```
require(
    gnosisSafe.getThreshold() == oldThreshold, "SecSCThresholdAction: WRONG_OLD_THRESHOLD"
);
```

and

```
require(
    gnosisSafe.getThreshold() == newThreshold, "SecSCThresholdAction: NEW_THRESHOLD_NOT_SET"
);
```

misspell the error message "SetSCThresholdAction" as "SecSCThresholdAction".

Code corrected:

Arbitrum Foundation has fixed the typos. The relevant code snippets look as follows:

```
require(constitution.constitutionHash() == newConstitutionHash, "NEW_CONSTITUTION_HASH_SET");
```

```
require(
    gnosisSafe.getThreshold() == oldThreshold, "SetSCThresholdAction: WRONG_OLD_THRESHOLD"
);
```

and

```
require(
    gnosisSafe.getThreshold() == newThreshold, "SetSCThresholdAction: NEW_THRESHOLD_NOT_SET"
);
```